SHOWME.how: Isolation, Interoperation, and Orchestration for FAIR Computational Research

Alan Correa [1], V Mithlesh Kumar [1], Anil Yildiz [1], Julia Kowalski [1]

[1] correa@mbd.rwth-aachen.de, yildiz@mbd.rwth-aachen.de
Chair of Methods for Model-based Development in Computational Engineering
RWTH Aachen University, Aachen, Germany

Abstract: Computational research in science and engineering requires integrating diverse methods, data formats, hardware platforms, and teams of collaborators with complementary expertise. The FAIR principles establish findability, accessibility, interoperability, and reusability as central goals for research outputs, and are widely adopted as criteria for good research practice. Yet the principles define goals without specifying how to achieve them given heterogeneity in the components required by a computational study. In practice, teams assemble ad-hoc approaches that grow fragile as components evolve, making reproducibility and reusability difficult to sustain. We present SHOWME.how, an approach built on three complementary pillars, Isolation, Interoperation, and Orchestration. Each software component is isolated with its dependencies in a self-contained environment; software components interoperate through language-agnostic data formats and standard transfer protocols; and we coordinate data movement, scheduling, and resource management through a scientific workflow management system. Together, the pillars allow computational studies to remain reproducible, portable, reusable, and extensible to new collaborators, methods, data, and infrastructure as they grow and evolve. We demonstrate the approach across three use cases spanning Bayesian calibration of a hemolysis model and a geophysical mass flow model, and benchmarking of Gaussian process emulators for high-dimensional problems.

Keywords: computational workflows, heterogeneity, reproducibility, FAIR, high-throughput computing, inverse problems, benchmarking

1 Introduction

Modern science and engineering has become highly computational, and increasingly interdisciplinary, hence collaborative by design. To answer a scientific question, solve an engineering problem, or develop a novel method or design, we routinely rely on data from experiments, field observations, and simulations, on computational models and algorithms implemented in several programming languages, on computational infrastructure ranging from laptops to high-performance computing (HPC) clusters and cloud, and on a diverse team of collaborators having experience in different domains, methods, and tools. In practice, a computational scientist who can develop and implement

computational models may work alongside a domain expert who has access to field data and application-specific knowledge, and a specialist in uncertainty quantification who has knowledge of probabilistic and statistical methods. In addition, a data scientist who manages and curates the data flowing through the study, and a research software engineer (RSE) with expertise in building reliable and maintainable research software, often completes the research team. This diversity of competences is intrinsic to real-world research and development teams and necessary for productive science and engineering. Yet the systems these teams build are not only difficult to assemble, but also tend to grow fragile over time, resulting in work that is hard to reproduce and hard to reuse.

In contrast to this observation, computational research has often been approached as a problem of assembling appropriate software components and data pipelines. Scientific workflow management systems (SWMSs), for instance, coordinate software and data flows [LHWF26, SCA+26], but we observe that they treat human collaborators as external operators rather than integral parts of the system. The computational research ecosystem (Figure 1), however, is where data, computational units, and human collaborators interact as integral parts of the same system. This broader view reveals that heterogeneity is amplified by the collaborators themselves, each bringing preferred tools, languages, and environments to the study.

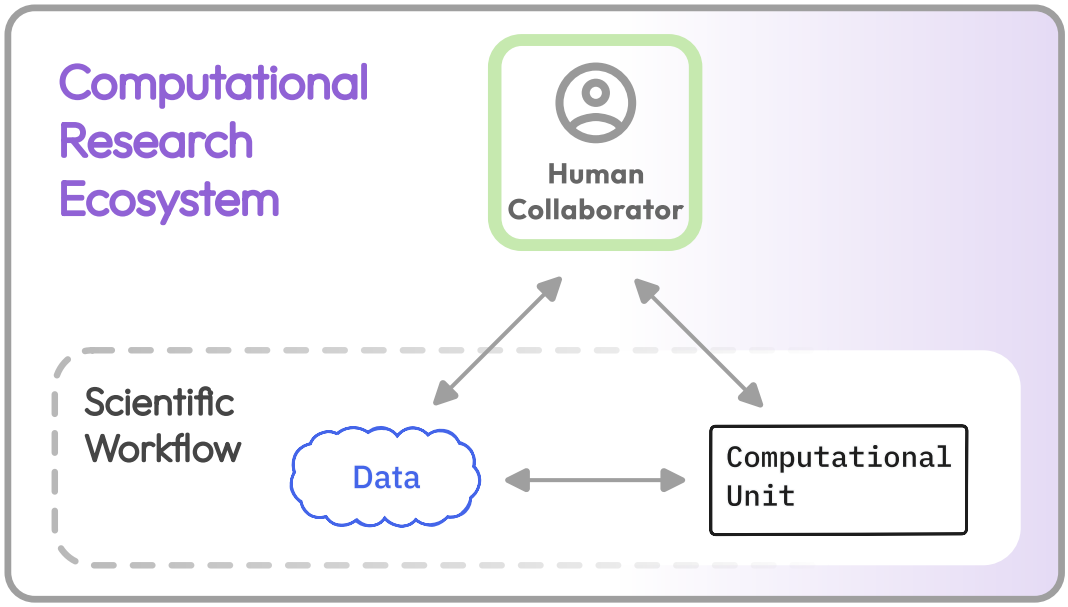

Figure 1: The computational research ecosystem. A scientific workflow (dashed box) orchestrates data and computational units but treats human collaborators as external. Optimal utilization, however, requires treating all three as integral parts. Visual encoding: computational units (black-bordered boxes), data (blue clouds), human collaborators (green rectangles).

The FAIR principles [WDA+16, BCK+22, WAB+25, GCS+20] define the attributes that scientific data, software, and workflows should exhibit, making findability, acces-

sibility, interoperability, and reusability central goals in scientific research. Yet while these goals are well established, a concrete path to achieve them in the face of heterogeneity across data, methods, computational infrastructure, and collaborators is far from straightforward. Having individually reproducible and reusable components does not guarantee a reproducible and reusable system; none of the FAIR properties is inherited once components are pieced together. For example, two components that each run reliably in isolation may still fail when connected if their compute environments conflict. Some teams even turn to SWMSs to build and operate these heterogeneous systems. However, these tools automate execution and scheduling but leave isolation and interoperation strategies to the team.

Taken together, these observations point to a deeper issue: modern computational research can no longer be reduced to ensuring the FAIRness of individual components or assembling software components and data pipelines. The system that emerges is more than the sum of its parts and tends to grow fragile as it evolves. Hence, we frame a computational study as a goal-oriented, strongly coupled sociotechnical system in which data, computational units, and human collaborators interact bidirectionally to generate value. This perspective reveals challenges that existing approaches do not address: FAIR properties are not inherited when individual components are pieced together, and existing tools leave the practical challenges of building and maintaining such computational studies to human collaborators.

We characterise the practical challenges that motivate the approach (Section 2.2) and survey the state of the art (Section 3). We then present SHOWME.how (Section 4), an approach built on three complementary pillars, Isolation, Interoperation, and Orchestration, that together allow a computational study to remain functional as it evolves. We apply the approach to three use cases spanning Bayesian calibration of a hemolysis model and a geophysical mass flow model, and benchmarking of Gaussian process emulators for high-dimensional problems (Section 5). We then show how the pillars operationalize robustness as the study evolves (Section 6). A gallery of use case implementations and living documentation is available at http://mbd-rwth.github.io/showmehow.

2 The Computational Research Ecosystem

Scientists and engineers conduct computational studies to answer research questions, develop models, and address engineering challenges. The computational research ecosystem supporting these studies is characterised by bidirectional interactions between three classes of components: data, computational units, and human collaborators (Figure 1). A computational unit couples a model or algorithm, in its concrete implementation, with the compute environment in which it runs. The compute environment has two layers. The software ecosystem covers the programming language and operating system required by the implementation. The hardware platform covers the physical or virtual resources, local or remote, on which the computational unit executes. Data comprises the inputs consumed by computational units, the outputs they produce, and the intermediate artefacts exchanged between them. Human collaborators are the researchers, engineers, and

domain experts who design and operate the study, and interpret its results. In principle this role could also be filled by AI agents, but we restrict our attention solely to human collaborators in this work.

The components of the computational research ecosystem are heterogeneous by nature. Data appears in many formats and structures, from geometries, boundary conditions, and observations as inputs, to quantities of interest, metrics, visualisations, and reports as outputs. These data may reside on local disks, in cloud object storage, or in data repositories. Models and algorithms span physics-based simulators and their surrogates, data-driven models, and frameworks for design, optimisation, inference, and analysis. Compute environments vary in both software ecosystem and hardware platform. The software ecosystem ranges across programming languages such as Python, Julia, R, C++, Fortran, and MATLAB, each typically dictated by the models and algorithms in use, and operating systems such as Linux, macOS, and Windows. The hardware platform ranges across local workstations, HPC clusters, and cloud, with accelerators such as GPUs alongside CPUs. This diversity stems from the dynamic nature of the computational research ecosystem and from the decisions made by human collaborators when designing computational studies. Domain specialists, methods experts, data scientists, and RSEs each bring different skills, training, and preferred toolsets to the study.

Figure 2 illustrates these components in a representative computational study: the Bayesian parameter calibration of a shallow mass flow model used in geophysical hazard simulation. This inverse problem is solved by chaining three computational units: a forward simulator, a surrogate model, and a calibration process, each developed and operated by collaborators with complementary expertise.

2.1 Recurring Workflow Patterns

Within the computational research ecosystem, human collaborators increasingly organise their computational studies using scientific workflows, in which we commonly encounter three recurring structural patterns (Figure 3). These patterns describe how computational units and data are coordinated; we introduce them here as they are referenced throughout the paper.

Pattern A: Parameter Sweep Pipeline In a computational study using a parameter sweep pipeline, a single computational unit processes many distinct input configurations. A Generator produces a large set of inputs (parameter sweeps, different initial conditions, or sampled configurations); each generated input is independently dispatched to a Method (a model or algorithm). The Method processes each input and produces an output; the outputs are collected by a Collector and aggregated into a dataset for downstream use. Common instances include training a surrogate model from a physics-based simulator using a space-filling sampling scheme that spreads samples across the relevant parameter ranges (see Figure 2), forward uncertainty propagation via Monte Carlo sampling where output statistics are derived from the ensemble, and Sobol sensitivity analysis where the full (input, output) dataset is processed to compute variance-based sensitivity indices.

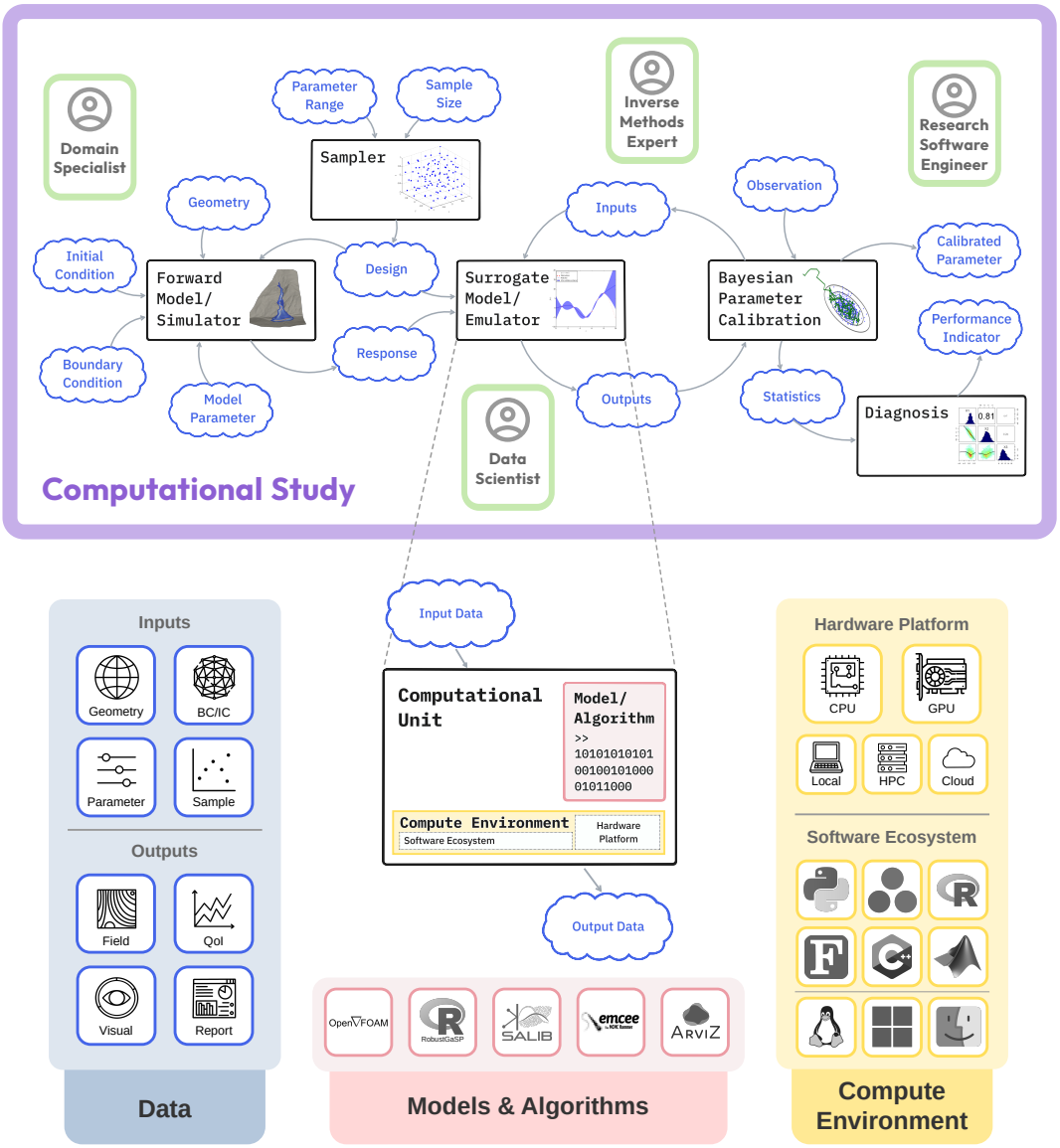

Figure 2: A computational study and the dimensions of its heterogeneity in data, computational units, and human collaborators. Top: a representative study in action: human collaborators (green rectangles) with different expertise coordinate computational units (black-bordered boxes) that consume and produce data (blue clouds) to carry out a research task. Bottom: the anatomy of a scientific workflow: a computational unit transforms input data into output data through a model or algorithm executed in a compute environment.

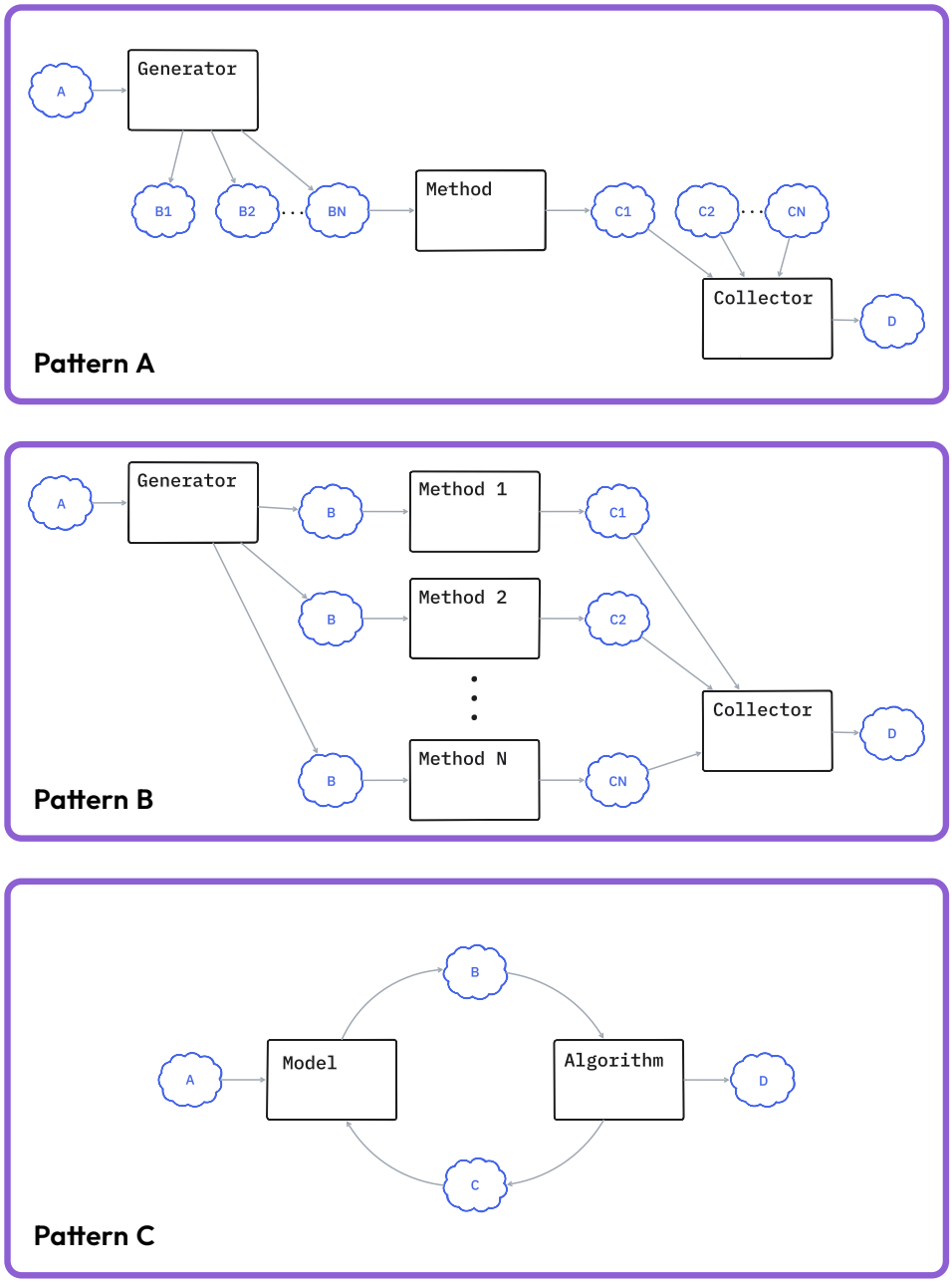

Figure 3: Three recurring workflow patterns in computational studies: Pattern A (parameter sweep pipeline), Pattern B (fan-out/fan-in pipeline), and Pattern C (model-based X in an online loop). Data is indicated via blue clouds and computational units via black-bordered boxes.

Pattern B: Fan-Out/Fan-In Pipeline In a computational study using a fan-out/fan-in pipeline, many distinct computational units process the same input data. A Generator produces a shared input that is dispatched to multiple Method implementations; each produces an output, and the outputs are gathered by a Collector for comparison. Herein, the Method implementations are of the same model or algorithm, differing in their mathematical formulation, programming language, or both. Common instances include method benchmarking, where competing implementations are evaluated against a shared input; ensemble runs with different random seeds, whose outputs are aggregated to characterise variability or assess convergence; and inference diagnostics, where shared inference results are fanned out to complementary analyses and gathered into a report.

Pattern C: Model-Based X in an Online Loop In a computational study using a model-based online loop, a model and an algorithm are coupled iteratively, exchanging data repeatedly until a desired objective is achieved. A Model and an Algorithm are coupled at run time as a client–server pair; the Algorithm drives the Model iteratively, requiring both to run simultaneously. Common instances include optimisation, where the model is queried iteratively to optimise an objective, inverse uncertainty quantification (UQ), such as parameter calibration (see Figure 2), model selection, and optimal experimental design, and data assimilation, where observations are used to update posterior estimates.

In any of these patterns, computational units can be instantiated in parallel when resources allow, reducing wall-clock time without changing the pattern structure. The patterns can also be composed. The Method in Pattern A or Pattern B may itself be a Pattern C, where a model and an algorithm are coupled in an online loop. Furthermore, a computational unit may itself house tightly coupled internal components, such as an MPI-parallel multi-physics solver or a co-simulation.

2.2 Challenges in the Computational Research Ecosystem

The computational research ecosystem is constantly evolving: new data sources become available, computational units are updated, replaced, or added, and human collaborators join or leave. Each such event is a potential point of failure. In practice, teams of human collaborators performing computational studies deal with the heterogeneity of the computational research ecosystem in one of two ways, each of which falls short as the ecosystem evolves.

Data feedforward. Human collaborators complete their part of the computation and manually transfer results to the next collaborator via shared drives or messaging. This allows each human collaborator to work with their preferred tools and workflows, at the cost of data versioning, traceability, and provenance. For studies following Pattern C (Section 2.1), such as Bayesian parameter calibration, where a UQ method must query a forward model iteratively throughout the inference process, the approach is not merely inefficient but physically infeasible.

Software wrappers. Human collaborators target composability of computational units through wrapper libraries, which expose models and algorithms implemented in different programming languages through a unified interface. The responsibility of writing and maintaining these wrappers falls on human collaborators. When an underlying library updates, the wrapper must follow; if it does not, the study either breaks or is locked to an older version of the model or algorithm. As a result, such wrappers frequently go stale. Moreover, adopting a newly published model or algorithm in a different programming language requires writing a new wrapper or forgoing it entirely.

The limitations of the above two approaches reveal deeper technical challenges that persist regardless of which is adopted, rooted in how computational units and data are managed within the computational research ecosystem.

Unit-level reproducibility. A reproducible computational study requires that each computational unit be individually reproducible. In practice, missing or incomplete environment specifications often make it impossible to rebuild the compute environment in which a unit was intended to run [CP16].

Dependency conflicts. Computational units developed independently sometimes require incompatible versions of the same library or runtime. Sharing a single environment, as in the software wrapper approach, makes these conflicts irreconcilable, forcing collaborators to compromise on versions or maintain fragile patches.

Environment drift. Even without deliberate changes, a computational unit that ran correctly months ago may silently fail today, as upstream packages are updated, system libraries change, and package registries remove older versions. Without explicit environment pinning, the software ecosystem drifts over time and results become irreproducible.

When human collaborators assemble computational units into a scientific workflow, further challenges emerge that become visible only from a systems perspective.

Language-tied data formats. Data is commonly serialised in formats tied to a specific language, for example .pkl or .npy in Python, or .jld2 in Julia. These are convenient within one programming language ecosystem but become barriers when a study needs to incorporate new human collaborators or computational units that require different programming languages, forcing data conversion or language adoption. Adopting a newly published model or algorithm compounds the problem, as it either cannot read existing data or requires a conversion step, adding friction precisely where rapid prototyping matters most.

Provenance and traceability. As data flows between computational units, tracking which version of which unit produced a given result becomes non-trivial. Without explicit provenance, debugging unexpected results requires reconstructing the chain of computation by hand, which is expensive when units involve long-running HPC jobs. More

broadly, without provenance, scientific claims produced by the study cannot be verified by reviewers or collaborators.

Platform portability. Human collaborators work on different operating systems — Linux, macOS, and Windows — meaning that units must remain functional across host operating systems that may resolve package dependencies differently or behave differently at runtime. Additionally, moving a computational study from a local workstation to an HPC cluster or cloud environment reveals assumptions baked into computational units about the host environment: container runtimes differ, schedulers vary, and what runs on a laptop may not run on HPC without unit-code changes. A further complication arises when different units must run on different infrastructure simultaneously, such as a computationally expensive model on HPC alongside a lighter unit running locally, requiring the study to coordinate across infrastructure boundaries at runtime.

Coordination overhead. As a computational study grows in the number of units and the complexity of their dependencies, manually tracking which units to run, in what order, and where their outputs go becomes error-prone and unsustainable. Interrupted runs compound the problem, requiring manual identification of completed steps and selective re-execution. Such a process is both fragile and time-consuming.

Human collaborator harmonization. Human collaborators occupy distinct roles: a domain specialist brings the problem and domain expertise, a methods expert brings the algorithmic solution, a data scientist manages, curates, and analyses the data flowing through the study, and an RSE builds and maintains the necessary infrastructure for the study. They also span different stages of experience, from master's students contributing individual units to principal investigators guiding the overall computational study. A shared vocabulary and clear input/output interfaces for each computational unit must be negotiated across all these roles and stages.

Together, these challenges make it difficult to build and sustain computational studies involving heterogeneous data, computational units, and human collaborators in a way that supports productive and efficient research.

3 State of the Art

A range of tools and standards address parts of the challenges described in Section 2.2. Each covers one or more dimensions of the problem but leaves others open.

Normative frameworks. The FAIR principles establish normative criteria for research outputs to be Findable, Accessible, Interoperable, and Reusable [WDA+16, BCK+22, GCS+20, WAB+25] across scientific data, software, and workflows, but do not prescribe how to achieve these properties when the components of a computational study are heterogeneous.

Software dependency management. Conda [con] and Mamba [QM] manage software dependencies by creating isolated, reproducible environments. Container tools such as Docker [Mer14] and Podman [HWB+18] extend this isolation to the operating system level, packaging a unit and its dependencies into a self-contained image. Together, these tools can make an individual unit reproducible. When a study involves multiple units with conflicting dependencies, however, sharing a single environment across the whole study is not viable; each unit requires its own isolated environment. None of these tools individually addresses how units with isolated environments should exchange data or be orchestrated together.

Scientific workflow management systems. Scientific workflow management systems coordinate execution and data flows across computational units. Suter et al. [SCA+26] provide a unified terminology, distinguishing scheduling models, environment integration, and data handling as key axes of variation; Nextflow [DCF+17] and Snakemake [MJL+21] are representative systems. For environment specification, SWMSs typically rely on conda or containers; Spack [GLC+15] is natively supported by Nextflow as an HPC-focused alternative, while Nix and Guix [VMT22] offer stronger reproducibility guarantees through cryptographic hashing of the full dependency graph but require custom tooling outside standard SWMS environment directives. Six SWMSs are evaluated in [DGL+23] against reproducibility and reuse criteria, finding per-process environment specification the most portable approach, but identifying an unresolved tradeoff between strongly-typed process interfaces and wrapper overhead that forces unit-code modification. Cohen-Boulakia et al. [CBC+17] evaluate SWMSs against four reproducibility levels and find that most fail to meet system-wide packaging criteria, cannot update broken environments without manual intervention, and require high technical skill to connect workflows across different systems; their analysis covers life-science pipelines where units share a common runtime and does not address settings where units differ in language, operating system, and infrastructure. In all cases, SWMSs leave it to the team to decide how units should be packaged and how data should be serialised at their boundaries.

Workflow platforms. Pegasus [DVJ+15] focuses on executing large-scale workflows across heterogeneous distributed infrastructure, handling job scheduling, data movement, and fault tolerance at scale. It treats each unit as a black box, making no prescription about how units should be packaged or what format data should take at their boundaries. The Galaxy Project [The24] enforces reproducibility through a managed tool ecosystem, primarily for bioinformatics; each tool must be registered in Galaxy's ToolShed with a descriptor specifying its dependencies and parameters, coupling reproducibility to Galaxy's own ecosystem. AiiDA [HZU+20] provides graph-database provenance tracking and automated workflow execution across multiple research domains; its provenance records capture all computational steps and data dependencies in a queryable graph, richer than the execution logs SWMSs natively produce, but it requires a dedicated message broker, a running database instance, and Python plugin wrappers for each code. ColonyOS [KBB+25] provides language-agnostic orchestration across geographically dis-

persed compute nodes through a centralised server that executors poll over HTTP, at the cost of a persistent broker and database that all participants must connect to. All four require either dedicated infrastructure or adoption of an existing tool ecosystem, limiting their accessibility for general-purpose computational studies.

Complementary standards. CWL [CAI+22] provides a vendor-neutral standard for portable workflow definitions, specifying what to run and how steps connect, but does not prescribe how units should be isolated from one another, how data should be serialised at unit boundaries, or how Pattern C (Section 2.1) should be handled when a unit runs as a long-lived service process.

Pursuing interdisciplinary computational research in a FAIR manner requires the ability to build and sustain computational studies that accommodate the heterogeneity of data, computational units, and human collaborators as the computational research ecosystem evolves. Existing approaches address either the reproducibility of individual units or the coordination across units and data, but none treats the computational research ecosystem as a whole. What we need is not more tools, but an approach that addresses all the identified challenges simultaneously.

4 SHOWME.how

We present SHOWME.how, an approach built on three complementary pillars: Isolation, Interoperation, and Orchestration (Figure 4), to address the challenges identified in Section 2.2. Each pillar addresses a distinct class of these challenges. A study can adopt any one of the pillars independently, adding the others as it grows and its requirements evolve.

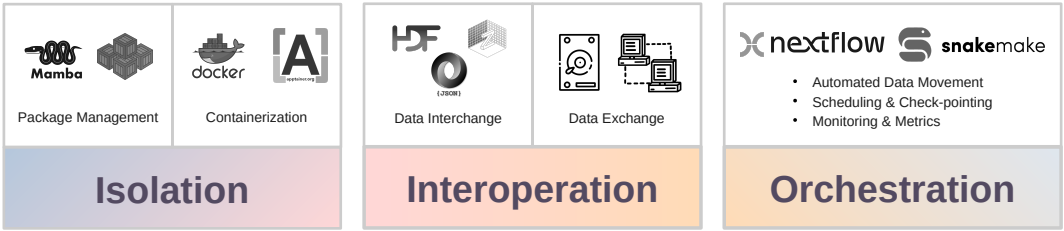

Figure 4: Overview of the SHOWME.how approach: Isolation, Interoperation, and Orchestration, with representative technologies for each pillar.

4.1 Isolation

The goal of Isolation is that any computational unit can be integrated into a computational study regardless of its programming language, operating system, and dependencies, without interfering with any other unit, so that each human collaborator can contribute and develop computational units in isolation. We can achieve this by encapsulating the

compute environment of each unit through package management and containerisation, so that units remain self-contained and portable. The right isolation strategy depends on the specific requirements of the study. The following questions help in identifying it:

1. What software ecosystem does each unit require, in terms of programming language, operating system, and dependencies?
2. Where will each unit run? On a collaborator's laptop, an HPC cluster, or a cloud environment?
3. What level of reproducibility is required? Is numerical equivalence across platforms sufficient, or is a fully controlled environment that also pins system libraries and floating-point behaviour needed?
4. How heterogeneous is the study in terms of human collaborators, operating systems, CPU architectures, and infrastructure, and how much change is expected?

Iso1 — Package management. Isolation at this level is achieved using a language-agnostic, cross-platform package manager. We recommend conda or mamba [con, QM], as they are natively supported by the SWMSs we discuss in Section 4.3. A unit's software dependencies are specified in an environment.yml file, but this alone does not guarantee reproducibility: package versions and all their indirect dependencies are not pinned, and some packages resolve differently across operating systems and CPU architectures. We therefore generate conda-lock files [con20], one per operating system and architecture, which pin every package to an exact version so that the software dependencies of a unit can be recreated identically on any matching platform. Per-OS, per-architecture locks are sufficient for numerically equivalent reproducibility, ensuring consistent results across the platforms each human collaborator uses. Other package managers such as pixi [AZV+] and the Julia package manager [BEKS17] have locking built in by default, but are not yet natively supported by the SWMSs we consider.

Iso2 — Containerisation. When package managers with pinned lock files are not sufficient, containerisation is the next step. This applies to computational units whose dependencies are specified through complex build systems, are not available on conda, or require legacy software and operating systems. A container packages not just the software dependencies but the entire operating system layer, so that the unit is fully independent of the host operating system. Each unit requires a Dockerfile as its build definition, which can then be built into a container image using Docker [Mer14], Podman [HWB+18], or any compatible container runtime. We recommend one container image per unit, so that a dependency change in one unit cannot affect any other. Container images are architecture-specific; when the study spans multiple CPU architectures, we need to build multi-architecture manifests that bundle architecture-specific images under a single tag, allowing the correct variant to be pulled automatically.[1]

[1] docker buildx is a Docker CLI plugin that enables building images for multiple CPU architectures from a single command, producing a multi-architecture manifest.

Iso3 — HPC. A Docker image is sufficient for laptop and cloud environments. HPC clusters typically restrict Docker for security reasons, making Apptainer [KSB17] the container runtime of choice. Apptainer can build images directly from an Apptainer definition file, but when the build definition requires frequent iteration, the lack of layer-based caching makes development slow, as every change triggers a full rebuild. We therefore recommend the pipeline Dockerfile → Docker image → Apptainer .sif image, converting the Docker image to an Apptainer image as a final step. This keeps a single build definition that benefits from Docker's layer caching during development, while producing an Apptainer image ready for HPC deployment.

Iso4 — Automation. Maintaining lock files across operating systems and architectures, and building container images across CPU architectures, is combinatorial work that does not scale when done manually. We recommend automating this process using continuous integration (CI). A matrix build can regenerate per-OS, per-architecture conda-lock files on every change to environment.yml, and build and publish multi-architecture container images whenever the Dockerfile changes. This shifts the maintenance burden from individual collaborators to a single automated pipeline.

For units where conda-lock alone is insufficient but a full Dockerfile is too heavy, a middle ground exists, achieved by wrapping a locked conda environment inside a minimal container image.[2] This combines conda's resolution of language-level dependencies with the container's control over the system layer, in a single artefact that can be built and automated in the same CI pipeline.

4.2 Interoperation

The goal of Interoperation is that data can flow between computational units regardless of the programming language each unit uses or the infrastructure on which it runs. We can achieve this by adopting language-agnostic data interchange formats and platform-agnostic data exchange protocols at unit boundaries. This directly addresses the Interoperability dimension of FAIR [WDA+16], which calls for data to use a broadly applicable language for knowledge representation, here realised through concrete format and protocol choices at unit boundaries. The right interoperation strategy depends on the specific requirements of the study. The following questions help in identifying it:

1. What is the nature of the data being exchanged? Is it tabular, hierarchical, or multi-dimensional arrays?

2. Do all units run locally, or does the study span multiple compute locations such as a laptop, an HPC cluster, or cloud infrastructure?

3. What is the pattern of data exchange? Does the study transfer data once per unit, or does it involve repeated exchanges during execution?

[2] Both SWMSs can also produce such images automatically. Snakemake's --containerize flag emits a single Dockerfile bundling a workflow's conda environments, and Nextflow's Wave service builds a container image on demand for each declared conda environment.

4. How might the interoperation needs of the study grow? Will the volume of data increase, or will the heterogeneity of units exchanging it?

Int1 — Data formats. All data at unit boundaries must use language-agnostic interchange formats, so that adding a unit in a new programming language never requires changes to existing units. Language-specific binary formats such as .pkl (Python), .npy (NumPy/Python), and .jld2 (Julia) should not appear at boundaries, as a human collaborator using R or C++ would need to install and maintain a Python or Julia runtime solely to read another unit's output. Text formats such as CSV and JSON are portable and an acceptable starting point when the data is not numerically heavy; they become a poor choice when the data consists primarily of floating-point values, as they are string-based, which introduces significant storage overhead and risks precision loss when insufficient digits are written.[3] For data consisting primarily of floating-point values, we recommend binary formats: HDF5 [The] or netCDF [NDR+] for hierarchical and structured data, Zarr [MK23] for multi-dimensional arrays, and Apache Parquet [Apand] for tabular data.

Int2 — Location. The exchange mechanism between computational units depends on where they execute. Filesystem exchange is the simplest starting point, with no network dependency; each unit writes its output to a file path and the next unit reads from it. This holds whether units execute on a single laptop or across nodes of an HPC cluster with a shared filesystem such as Lustre or GPFS. On Windows and macOS, Docker containers run inside a Linux virtual machine and cannot directly access the host's interprocess communication mechanisms; filesystem exchange and network protocols are the only reliable options for same-machine exchange between a containerised unit and a non-containerised one. We recommend starting with filesystem exchange and switching to remote storage only when the study spans separate infrastructures with no shared filesystem between them, such as a laptop and a cloud instance or two separate HPC systems. S3-compatible object storage suits this case well, as it is accessible from any infrastructure via HTTP and requires no shared mount. When the source or destination exposes a filesystem rather than an object store, such as a remote workstation or an existing dataset on a server, SFTP is more appropriate.

Int3 — Bandwidth. Bandwidth governs how much data can flow across computational units per unit time. It becomes the limiting factor in two situations: when units exchange large volumes of data in bulk over a network; and when units exchange data at high frequency, as in Pattern C (Section 2.1), where one unit queries another repeatedly during execution. For bulk transfers, we recommend Zarr [MK23] over HDF5; unlike HDF5, Zarr is designed for chunked access over HTTP and S3, making it well-suited to large array data moving across a network or object storage. For high-frequency ex-

[3] A 64-bit double requires 17 significant decimal digits for a guaranteed round-trip, leading to string representations of 20–24 ASCII characters (160–192 bits) versus 64 bits in binary (a bloat factor of approximately 2.5–3×).

change, we recommend an HTTP-based client–server approach, where one unit acts as a server exposing its computation and the other acts as a client that queries it repeatedly. The approach requires choosing a transport protocol and a serialization format. HTTP provides a simple, cross-language point-to-point interface with broad support. For the serialization format, JSON is acceptable when payloads are small, but its overhead grows with array size and can become a bottleneck for large numerical payloads; binary formats such as MessagePack [mes] or Protocol Buffers [Pro08] impose substantially lower serialisation overhead. UMBridge [SCD+23] provides a ready-made implementation of this approach for model-based uncertainty quantification; it uses HTTP and JSON and requires no manual data transfer, but its interface accepts only array-based inputs and outputs, making it unsuitable when units need to exchange data that is not array-based.

As a rule of thumb, start with CSV or JSON and filesystem exchange; move to binary formats such as HDF5, Zarr, or Parquet as the data becomes primarily floating-point. For the exchange mechanism, switch to remote storage when the study spans separate infrastructures; adopt an HTTP-based client–server approach only when units need to exchange data at high frequency during execution.

4.3 Orchestration

The goal of Orchestration is that the study runs reliably end-to-end, even as units are added, collaborators change, or the infrastructure it runs on evolves. We can achieve this by using a SWMS to automate scheduling, data movement, and checkpointing, so the study can operate seamlessly as its needs grow. The right orchestration strategy depends on the specific requirements of the study. The following questions help in identifying it:

1. Is the study complex enough to warrant a SWMS? How many units are there, and how complex are their dependencies?
2. Does the study need to execute across different infrastructure, such as running some units locally and others on an HPC cluster or cloud, and what schedulers or executors does that require?
3. Would restarting the study from scratch on failure be costly, either because individual units are long-running or because the study involves a large number of unit executions?
4. Does the study require any unit to run as a long-lived service process, as in Pattern C, rather than completing after a single execution?

Orch1 — Automation. In a computational study with few computational units and human collaborators that runs once, execution can be coordinated with a script. A SWMS becomes necessary when the study needs to run repeatedly, when units span different environments or infrastructure, or when manual data movement and environment activation becomes a bottleneck.

We use Nextflow [DCF+17] and Snakemake [MJL+21] as representative examples throughout this section; SHOWME.how is not tied to any particular manager, and a

broader survey of the landscape is available in Suter et al. [SCA+26] and [DGL+23]. Nextflow runs on Linux and macOS (with WSL on Windows); Snakemake is cross-platform. Workflow extensions in Nextflow require Groovy and the JVM; Snakemake extensions are written in Python. Both managers share a similar unit contract, where each unit declares its inputs, outputs, script, and compute environment.[4] This contract ties directly to both preceding pillars; the environment declaration activates the isolation strategy defined for each unit, and the input/output declarations specify where and how data crosses unit boundaries. The contract also serves as the shared interface through which human collaborators with different roles can understand and contribute to individual computational units without needing to understand the implementation of every other computational unit in the study.

Orch2 — Scheduling. Scheduling determines where and how units execute. Both Nextflow and Snakemake follow a static planning approach [SCA+26], resolving the full dependency graph before execution begins and submitting each unit only when its declared inputs are available. They differ in how dependencies are expressed; Nextflow uses a dataflow paradigm where units are connected by explicit data channels, and Snakemake uses a file-based paradigm where dependencies are inferred from input and output file paths. A SWMS submits each unit to the appropriate executor automatically, whether a local process on a laptop, a SLURM job on an HPC cluster, or a cloud executor. In most cases, moving the study to a different infrastructure requires no changes to unit code; only the configuration specifying where and how units execute changes.

Orch3 — Incremental execution. Incremental execution means the SWMS tracks which units have completed successfully and skips them on rerun, so a failure mid-run restarts from the last completed unit rather than from scratch. This becomes essential when individual units are long-running or the study must execute a large number of them. Snakemake tracks completion through five default rerun-triggers: modification time and input files are tracked by file modification time, while code, parameters, and software environment are tracked by a hash; a unit is rerun whenever any of these change, and re-running the same command resumes automatically. Nextflow achieves the same through a work directory with per-task hashes that incorporate task inputs, script content, software environment, container, and a session identifier; re-running the same command starts fresh because the session identifier changes, so the –resume flag is required to resume from cached results. The work directory accumulates over time and requires periodic cleanup with nextflow clean. Beyond that, neither manager tracks changes to resources not declared as workflow inputs, and incremental execution assumes deterministic outputs. Both managers also capture basic execution provenance. Snakemake generates an HTML report with DAG visualisation, runtime statistics, and input/output records; Nextflow supports provenance through the nf-prov plugin [nex24], which produces records in RO-Crate [SSC+22] or PROV [LSM+13] format but requires explicit setup.

[4] Snakemake's native container runtime is Apptainer; Docker images declared in the environment are converted automatically.

Orch4 — Service processes. Pattern C (Section 2.1) requires a unit to run as a long-lived service process, specifically a model server that must remain alive while a client unit queries it repeatedly. Scientific SWMSs assume each unit starts, runs, and exits; a service process breaks this assumption. In Snakemake, a unit can be declared as a service rule; it is started before dependent units and terminated automatically once all consumers have finished. Nextflow has no native support for service processes. The common workaround is a file-based sentinel, where the server writes a file when it is ready and the client unit declares that file as an input dependency. The workaround has two failure modes. If the server crashes before writing the sentinel, the client unit never starts and the workflow stalls; if the server crashes after the client has started, the client cannot reach the server's URL and port and crashes with a connection error. Studies using Snakemake can use native service rules; studies using Nextflow can rely on the file-based sentinel workaround.

5 Use Cases

We applied the SHOWME.how approach to three studies spanning biomedical engineering, geophysical mass flow simulation, and high-dimensional surrogate modelling, collectively covering all three patterns from Section 2.1. The three studies are illustrative rather than exhaustive, but representative of the approach in practice.

5.1 Bayesian Calibration of a Hemolysis Model

In this study, we infer the parameters of a computational hemolysis model [DB26] that predicts red blood cell damage under mechanical stress, using Bayesian calibration against experimental measurements. Three collaborators with complementary expertise contribute to the study: a domain expert using macOS and familiar with Python, who has access to experimental data and can assess whether results are physically plausible; a UQ-methods expert on Windows in Python, who designs the calibration study and assesses whether the results are meaningful; and an RSE on Linux working across multiple languages, who builds and maintains the study and supports experiments. The study consists of a forward model, an MCMC unit, and a report unit. The forward model takes control variables and design parameters as inputs and returns a predicted hemolysis index; it is available in both Python and Julia, selectable at runtime. The MCMC unit queries the forward model in an online loop to infer model parameters (Pattern C). The report unit compiles results and diagnostics from the MCMC outputs. The study can run a single calibration or fan out across a sweep of experimental datasets (Pattern A) or across competing hemolysis model implementations (Pattern B), applying the same calibration setup to each variant.

The computational units are isolated in dedicated conda environments (Iso1); the Julia forward model is instead containerised in Docker (Iso2), as it cannot be managed through conda. A container definition file that mimics the Dockerfile creates the Apptainer image for HPC deployment (Iso3), so porting requires no modifications to unit code. Experimental data are fetched from a remote repository at setup time and prepro-

cessed to CSV; MCMC outputs are written in NetCDF using the ArviZ interface. Both are language-agnostic formats exchanged through the filesystem, so any unit can consume results regardless of language (Int1, Int2). The MCMC client queries the forward model via the UMBridge interface, so the two units exchange data without any language bridging (Int3). Each unit declares its inputs, outputs, script, and environment as a self-contained contract; the full workflow is automated in Nextflow (Orch1), which limits concurrency based on available resources when running calibrations in parallel (Orch2) and skips completed runs on rerun (Orch3). The forward model server writes a sentinel file containing its network port to a shared directory; the MCMC client starts and waits until the file is present before connecting (Orch4).

We ported the study to HPC by updating the Nextflow executor profile and building Apptainer images from the container definition files, with no modifications to unit code. The Julia forward model was ported later by the RSE, yielding a 10–100× speedup over the Python baseline. Diagnostic units were added after the study was first assembled without modifying any existing unit. Conda lock files and CI automation (Iso1, Iso4) remain to be added before wider distribution. In this study, human collaborator harmonization, platform portability, and coordination overhead, among others, were addressed through Iso1–3, Int1–3, and Orch1–4. The study is available at https://github.com/nicodirkes/bpc_hemolysis.

5.2 Bayesian Calibration of a Mass Flow Model

In this study, we use Bayesian inference with MCMC against field observations to calibrate the parameters of a geophysical mass flow simulator [ZYBK23, MPK+25, OWF+26] that models natural hazards such as landslides and avalanches. Because the forward simulator is computationally expensive, calibration proceeds in two stages: first a surrogate is trained on simulator runs, then MCMC inference proceeds using the surrogate. Two collaborators with complementary expertise developed the study: a UQ-methods researcher on Windows, proficient in Python and R, who wanted to apply surrogate-based calibration to a real-world use case that requires an expensive simulator; and a computational scientist with experience in high-fidelity mass flow simulators, who is also an RSE on Linux, and builds and maintains the study. The study spans four units across three language ecosystems: a Latin Hypercube Sampling unit in Julia generates a parameter design; the forward simulator in Python evaluates each input configuration from the design space and returns model responses; a surrogate training unit in R fits a Gaussian process emulator to the design and response (Pattern A). A surrogate serving unit in R then provides access to predictions over HTTP, and an MCMC inference unit in Python queries it in an online loop to infer the model parameters (Pattern C).

Each unit is isolated in a dedicated conda environment (Iso1) and additionally containerised using Apptainer for running on HPC (Iso2, Iso3), so no unit code needs to be modified. Latin Hypercube samples and forward model responses are exchanged as CSV files; MCMC outputs are written in HDF5. Both are language-agnostic formats exchanged through the filesystem (Int1, Int2), bridging the Julia, Python, and R units across the pipeline. The surrogate unit is decomposed into a training sub-process and

a serving sub-process so that the emulator need not be retrained on every run; they exchange the trained model as an RDS file. The surrogate server provides access to predictions via the UMBridge interface; the Python MCMC client queries it without any language bridging (Int3). The full workflow is automated in Nextflow (Orch1), which sequences the two stages so that MCMC calibration starts only after the surrogate is trained (Orch2) and skips completed computational model simulations on rerun, so an interrupted run need not repeat the expensive simulation stage (Orch3). The surrogate server signals readiness by writing its network port to a sentinel file; the MCMC client connects once the file appears (Orch4).

Assembling this study is where we first identified the sentinel file pattern (Orch4). An R server implementation of the UMBridge interface was not yet available; we implemented it locally, and an upstream contribution is underway. We have executed the study end-to-end on Windows and Linux laptops and on a local Linux cluster. In this study, dependency conflicts, coordination overhead, and human collaborator harmonization, among others, were addressed through Iso1–3, Int1–3, and Orch1–4. The study is available at https://github.com/thealanjason/bpc_massflow.

5.3 Benchmarking Gaussian Process Emulators for High-Dimensional Problems

In this study, we benchmark Gaussian process emulator (GPE) implementations against five benchmark datasets, with a focus on high-dimensional inputs and outputs. The study was developed by a master's student on macOS, proficient in Python, who needed to apply a GPE method from other domains to the geohazard domain and benchmark it against existing implementations in a reproducible way. The study fans out these datasets to multiple emulator implementations in parallel, then aggregates and scores the results (Pattern B). The datasets span synthetic and real-world scenarios, generated locally or fetched from Figshare and Zenodo; all are preprocessed by a shared preprocessing unit. The emulator implementations span two language ecosystems, with twelve implementations built on GPyTorch [GPB+18] in Python and RobustGaSP [GWB18] in R, covering exact inference, deep kernel learning, dimensionality reduction, and multi-output methods. Each emulator unit reads the preprocessed data, runs independently, and writes results; a metrics aggregation unit gathers all outputs and scores them for comparison.

Each emulator unit runs in its own conda environment with a pinned lock file (Iso1), so adding a new emulator introduces no dependency conflict with any existing unit. Data at all unit boundaries is in HDF5, exchanged through the filesystem (Int1, Int2), so any emulator reads and writes through the same format regardless of language. The study is automated in both Nextflow and Snakemake (Orch1); both support local and SLURM execution profiles (Orch2), and the Snakemake version skips completed emulator runs on rerun (Orch3). CI automatically regenerates conda lock files whenever an environment specification changes (Iso4), and a benchmark run is triggered on every push to the main branch with results uploaded as a GitHub artifact.

The master's student applied kPCA-PPGaSP, a GPE method from the literature, as a new unit and compared it against other implementations; adding the new unit

required no changes to any existing unit. We implement the study in both Nextflow and Snakemake to demonstrate that our approach is not tied to any particular workflow manager; both versions follow the same unit contracts and produce consistent results. GPU-accelerated execution is supported for PyTorch-based emulators in the SLURM profile. We have run the study on macOS and on a local Linux cluster. In this study, dependency conflicts, environment drift, and unit-level reproducibility, among others, were addressed through Iso1, Iso4, Int1–2, and Orch1–3. The study is available in two versions: Nextflow at https://github.com/gary8564/gpe_bench_nxf and Snakemake at https://github.com/gary8564/gpe_bench_smk.

6 Robustness in the Computational Research Ecosystem

Productive research in science and engineering depends on building and maintaining robust computational studies as the study grows and changes. Drawing on the notion of robustness in sociotechnical systems [Glu12], a computational study is robust if it continues to produce reliable and accurate results in the face of disturbances. Just as FAIR properties are not inherited when components are assembled, neither is robustness — it must be deliberately engineered into the study. We operationalize robustness through four capabilities:

1. Tolerate perturbations: remain functional when individual computational units are updated or replaced.

2. Handle variability: remain effective across diverse operating systems, programming languages, hardware platforms, and data formats.

3. Withstand adverse conditions: remain operational under infrastructure migration and human collaborators joining or leaving, and recoverable on interruption of a long-running computation.

4. Support modular extensibility: accommodate new computational units and new human collaborators added after the study is first assembled, without breaking existing ones. Studies grow as new scientific questions are asked and new methods and data sources become available; a robustness framing must account for it.

The three pillars of SHOWME.how were designed to address the failure modes presented in Section 2.2 — component-level reproducibility, system-level coordination, and human collaborator harmonization. Each pillar addresses one or more of these capabilities, though none is secured by a single pillar alone. Isolation mainly underpins the ability to tolerate perturbations, Interoperation the ability to handle variability across languages and platforms, and Orchestration the ability to withstand adverse conditions and to support modular extensibility. The criteria in Table 1 make that coverage explicit and checkable against a concrete study.

This rubric is prescriptive: it states what a robust study must satisfy, not what any particular study has been shown to satisfy. The hemolysis use case (Section 5) satisfies

Table 1: Robustness checklist for a computational study. A study satisfying all criteria continues to produce reliable and accurate results under a wide range of practical conditions and perturbations.

Capability	Criterion	Strategy
Tolerate perturbations	Per-OS, per-architecture lock file or container image per unit.	Iso1–Iso2
	All random state seeded per thread, rank, and library.†	—
Handle variability	No language-tied serialisation format at any unit boundary.	Int1
	Workflow runs without unit-code changes across all target operating systems.	Iso1–Iso3
	If needed, live-loop coupling (Pattern C) uses a language-agnostic protocol.	Int3
Withstand adverse conditions	A single Dockerfile drives both Docker and Apptainer images; porting to HPC requires no unit-code changes.	Iso3
	Executor configuration handles scheduler and runtime differences.	Orch2
	CI is implemented to keep multi-architecture environment lock files and container builds current.	Iso2, Iso4
	The study resumes from the last completed step on interruption.	Orch3
Support modular extensibility	Each unit declares inputs, outputs, script, and environment as a self-contained contract.	Orch1
	No unit bridges programming languages through wrapper code.	Iso1–Iso2, Int1
	Adding a new computational unit requires no changes to any existing one.	Orch1, Iso1–Iso2, Int1
	A new human collaborator can reproduce the study from a repository clone, a single environment for the workflow manager, and one run command.	Iso1–Iso2, Orch1
	Any human collaborator can understand the purpose and interfaces of each computational unit from the unit contract alone.	Orch1, Iso1–Iso2, Int1

† A single global seed is insufficient under parallel execution; each thread, rank, and library maintaining independent state must be seeded separately (e.g. NumPy's Generator, Julia's Random.seed!, R's set.seed each manage independent state).

several criteria: the study runs end-to-end without unit-code changes on Linux, macOS, and Windows, and diagnostic units were added after the study was first assembled without modifying existing ones. All boundary files use language-agnostic formats (HDF5, NetCDF, CSV, or JSON), and a new human collaborator reproduced the study with a git clone, a single conda environment for the workflow manager, and one run command. Numerical equivalence of outputs across operating systems has not been verified systematically; the study involves stochastic units, and cross-OS equivalence is contingent on the random-state seeding criterion being fully implemented, which remains open. Nevertheless, the rubric remains useful independently of any particular use case: it provides a concrete set of criteria for assessing and improving robustness in any computational study operating within a heterogeneous and evolving computational research ecosystem.

7 Conclusion and Outlook

We examined the computational research ecosystem and framed a computational study as a goal-oriented, strongly coupled sociotechnical system in which data, computational units, and human collaborators interact bidirectionally to generate value. This new perspective revealed a set of practical challenges that stem from looking at the system as a whole: language-tied data formats, provenance, and portability across hardware platforms and operating systems; coordination overhead; and the harmonisation of human collaborators across distinct roles and experience levels. Existing approaches address either the reproducibility of individual units or the coordination across units and data, but none treats the computational research ecosystem as a whole and none ensures that FAIR properties are inherited when components are assembled into a study. What is needed is not more tools, but an approach that addresses all these challenges simultaneously. SHOWME.how addresses this gap through three complementary pillars: Isolation, Interoperation, and Orchestration, designed to manage heterogeneity in data, computational units, and human collaborators as computational studies evolve, with reproducibility and reusability following as consequences rather than standing as the sole goals.

Computational studies built following the SHOWME.how approach are reproducible, portable across hardware platforms and operating systems, reusable, and extensible to include new collaborators, methods, data, and infrastructure. Any computational unit can be integrated into the study regardless of the language it is written in or the operating system it runs on; data flows across unit boundaries in language-agnostic formats; and individual units can be carried into new studies without modification. We demonstrated the approach across three use cases spanning Bayesian calibration of a hemolysis model and a geophysical mass flow model, and benchmarking of Gaussian process emulators for high-dimensional problems, between them exercising all three recurring workflow patterns. We further showed that the three pillars operationalize robustness, allowing computational studies to tolerate perturbations, handle variability, withstand adverse conditions, and support modular extensibility (Section 5). The approach demonstrates that it works across diverse domains, team compositions, and workflow patterns.

As future work, several directions are worth pursuing. The robustness rubric provides

a prescriptive checklist; quantitative scoring and extension to new use cases call for further work. Streaming and digital-twin studies, where a model receives continuous live input, would require adopting SWMSs designed for streaming execution, such as those catalogued by the Workflows Community Initiative [SCA+26]. Finally, studies employing SHOWME.how already make explicit the unit contracts — the input, output, script, and environment specifications of each unit — that portable workflow descriptions require, making it straightforward to describe studies in CWL and package them as RO-Crates for sharing through WorkflowHub [SSC+22, GWB+25].

Acknowledgements: Financial support from the Deutsche Forschungsgemeinschaft (DFG) through grant IRTG-2379 is gratefully acknowledged. The authors thank Nico Dirkes, Gary Chang, and all collaborators whose studies are presented as use cases in this paper.

Bibliography

[Apand] Apache Software Foundation. Apache Parquet. https://parquet.apache.org/, n.d.

[AZV+] R. Arts, B. Zalmstra, W. Vollprecht, T. de Jager, N. Morcotilo, J. Hofer. pixi.
https://github.com/prefix-dev/pixi/releases/tag/v0.70.2

[BCK+22] M. Barker, N. Chue Hong, D. Katz, A.-L. Lamprecht, C. Martinez-Ortiz, F. Psomopoulos, J. Harrow, L. Castro, M. Gruenpeter, P. Martinez, T. Honeyman. Introducing the FAIR Principles for research software. Scientific Data 9:622, 2022.
doi:10.1038/s41597-022-01710-x

[BEKS17] J. Bezanson, A. Edelman, S. Karpinski, V. B. Shah. Julia: A fresh approach to numerical computing. SIAM Review 59(1):65–98, 2017.
doi:10.1137/141000671

[CAI+22] M. R. Crusoe, S. Abeln, A. Iosup, P. Amstutz, J. Chilton, N. Tijanić, H. Ménager, S. Soiland-Reyes, B. Gavrilović, C. Goble, The CWL Community. Methods Included: Standardizing Computational Reuse and Portability with the Common Workflow Language. Communications of the ACM 65(6):54–63, 2022.
doi:10.1145/3486897
https://doi.org/10.1145/3486897

[CBC+17] S. Cohen-Boulakia, K. Belhajjame, O. Collin, J. Chopard, C. Froidevaux, A. Gaignard, K. Hinsen, P. Larmande, Y. L. Bras, F. Lemoine, F. Mareuil, H. Ménager, C. Pradal, C. Blanchet. Scientific Workflows for Computational Reproducibility in the Life Sciences: Status, Challenges and Opportunities.

Future Generation Computer Systems 75:284–298, 2017.
doi:10.1016/j.future.2017.01.012

[con] conda contributors. conda: A system-level, binary package and environment manager running on all major operating systems and platforms.
https://github.com/conda/conda

[con20] conda-lock contributors. conda-lock: Reproducible lockfiles for conda environments. 2020.
https://github.com/conda/conda-lock

[CP16] C. Collberg, T. A. Proebsting. Repeatability in Computer Systems Research. Communications of the ACM 59(3):62–69, 2016.
doi:10.1145/2812803

[DB26] N. Dirkes, M. Behr. A Practical Computational Hemolysis Model Incorporating Biophysical Properties of the Red Blood Cell Membrane. 2026.
https://arxiv.org/abs/2601.19994

[DCF+17] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, C. Notredame. Nextflow enables reproducible computational workflows. Nature Biotechnology 35:316–319, 2017.
doi:10.1038/nbt.3820

[DGL+23] P. Diercks, D. Gläser, O. Lünsdorf, M. Selzer, B. Flemisch, J. F. Unger. Evaluation of tools for describing, reproducing and reusing scientific workflows. ING.GRID 1(1), 2023.
doi:10.48694/inggrid.3726

[DVJ+15] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, K. Wenger. Pegasus, a Workflow Management System for Science Automation. Future Generation Computer Systems 46:17–35, 2015.
doi:10.1016/j.future.2014.10.008

[GCS+20] C. Goble, S. Cohen-Boulakia, S. Soiland-Reyes, D. Garijo, Y. Gil, M. R. Crusoe, K. Peters, D. Schober. FAIR Computational Workflows. Data Intelligence 2(1–2):108–121, 2020.
doi:10.1162/dint_a_00033

[GLC+15] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, S. Futral. The Spack Package Manager: Bringing Order to HPC Software Chaos. In SC15: International Conference for High-Performance Computing, Networking, Storage and Analysis. Pp. 1–12. IEEE Computer Society, Nov. 2015.
doi:10.1145/2807591.2807623

[Glu12] O. Gluchshenko. Definitions of Disturbance, Resilience and Robustness in ATM Context. Technical report, December 2012.
https://elib.dlr.de/79571/

[GPB+18] J. R. Gardner, G. Pleiss, D. Bindel, K. Q. Weinberger, A. G. Wilson. GPyTorch: Blackbox Matrix-Matrix Gaussian Process Inference with GPU Acceleration. In Proceedings of the 32nd International Conference on Neural Information Processing Systems. Pp. 7587–7597. Curran Associates Inc., 2018.

[GWB18] M. Gu, X. Wang, J. O. Berger. Robust Gaussian Stochastic Process Emulation. The Annals of Statistics 46(6A):3038–3066, 2018.
doi:10.1214/17-AOS1648

[GWB+25] O. J. R. Gustafsson, S. R. Wilkinson, F. Bacall, S. Soiland-Reyes, S. Leo, L. Pireddu, S. Owen, N. Juty, J. M. Fernández, T. Brown, H. Ménager, B. Grüning, S. Capella-Gutierrez, F. Coppens, C. Goble. WorkflowHub: a registry for computational workflows. Scientific Data 12:837, 2025.
doi:10.1038/s41597-025-04786-3

[HWB+18] M. Heon, D. Walsh, B. Baude, U. Mohnani, A. Cui, T. Sweeney, G. Scrivano, C. Evich, V. Rothberg, M. Trmač, J. Honce, Q. Wang, L. Mandvekar, A. Reber, E. Santiago, S. Grunert, N. Dahyabhai, A. Bjorklund, K. Kushwaha, S. A. Sha, Y. Pu, M. Vasek, Podman Community. Podman: A tool for managing OCI containers and pods. Jan. 2018.
doi:10.5281/zenodo.4735633

[HZU+20] S. P. Huber, S. Zoupanos, M. Uhrin, L. Talirz, L. Kahle, R. Häuselmann, D. Gresch, T. Müller, A. V. Yakutovich, C. W. Andersen, F. F. Ramirez, C. S. Adorf, F. Gargiulo, S. Kumbhar, E. Passaro, C. Johnston, A. Merkys, A. Cepellotti, N. Mounet, N. Marzari, B. Kozinsky, G. Pizzi. AiiDA 1.0, a Scalable Computational Infrastructure for Automated Reproducible Workflows and Data Provenance. Scientific Data 7:300, 2020.
doi:10.1038/s41597-020-00638-4

[KBB+25] J. Kristiansson, U. Bodin, C. Borngrund, J. Delsing, J. Martinsson. ColonyOS: A Meta-OS for Computing Continuums. In 2025 IEEE International Conference on Cloud Engineering (IC2E). Pp. 236–247. 2025.
doi:10.1109/IC2E65552.2025.00041

[KSB17] G. M. Kurtzer, V. Sochat, M. W. Bauer. Singularity: Scientific Containers for Mobility of Compute. PLOS ONE 12(5):e0177459, 2017.
doi:10.1371/journal.pone.0177459

[LHWF26] U. Leser, M. Hilbrich, S. R. Wilkinson, R. Ferreira da Silva (eds.). Workflow Systems for Large-Scale Scientific Data Analysis. Berlin Universities Publishing, 2026.
doi:10.14279/depositonce-24000

[LSM+13] T. Lebo, S. Sahoo, D. McGuinness, K. Belhajjame, J. Cheney, D. Corsar, D. Garijo, S. Soiland-Reyes, S. Zednik, J. Zhao. PROV-O: The PROV Ontology. W3C recommendation, World Wide Web Consortium (W3C), 2013. https://www.w3.org/TR/prov-o/

[Mer14] D. Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. Linux Journal 2014(239):2, 2014.

[mes] MessagePack: It's like JSON. but fast and small. https://msgpack.org/. Accessed: 2026-06-10.

[MJL+21] F. Mölder, K. P. Jablonski, B. Letcher, M. B. Hall, C. H. Tomkins-Tinch, V. Sochat, J. Forster, S. Lee, S. O. Twardziok, A. Kanitz, A. Wilm, M. Holtgrewe, S. Rahmann, S. Nahnsen, J. Köster. Sustainable data analysis with Snakemake. F1000Research 10:33, 2021.
doi:10.12688/f1000research.29032.2

[MK23] J. Moore, S. Kunis. Zarr: A Cloud-Optimized Storage for Interactive Access of Large Arrays. In Proceedings of the Conference on Research Data Infrastructure. Volume 1. 2023.
doi:10.52825/cordi.v1i.285

[MPK+25] M. Mergili, H. Pfeffer, A. Kellerer-Pirklbauer, C. Zangerl, S. P. Pudasaini. r.avaflow v4, a multi-purpose landslide simulation framework. Geoscientific Model Development 18(23):9879–9896, 2025.
doi:10.5194/gmd-18-9879-2025

[NDR+] NSF Unidata, G. Davis, R. Rew, D. Heimbigner, E. Hartnett, W. Fisher, Others. NetCDF-C.
doi:10.5065/D6H70CW6
https://github.com/Unidata/netcdf-c

[nex24] nextflow-io contributors. nf-prov: Provenance plugin for Nextflow. 2024. Accessed: 2026-06-10.
https://github.com/nextflow-io/nf-prov

[OWF+26] F. Oesterle, A. Wirbel, J.-T. Fischer, A. Huber, P. Spannring. AvaFrame. 2026.
doi:10.5281/zenodo.4721446

[Pro08] Protocol Buffers Contributors. Protocol Buffers. https://protobuf.dev, 2008.

[QM] QuantStack, Mamba Contributors. mamba.
https://github.com/mamba-org/mamba

[SCA+26] F. Suter, T. Coleman, İ. Altinţas, R. M. Badia, B. Balis, K. Chard, I. Colonnelli, E. Deelman, P. Di Tommaso, T. Fahringer, C. Goble, S. Jha, D. S. Katz, J. Köster, U. Leser, K. Mehta, H. Oliver, J.-L. Peterson, G. Pizzi,

L. Pottier, R. Sirvent, E. Suchyta, D. Thain, S. R. Wilkinson, J. M. Wozniak, R. Ferreira da Silva. A Terminology for Scientific Workflow Systems. Future Generation Computer Systems 174:107974, 2026.
doi:10.1016/j.future.2025.107974

[SCD+23] L. Seelinger, V. Cheng-Seelinger, A. Davis, M. Parno, A. Reinarz. UM-Bridge: Uncertainty quantification and modeling bridge. Journal of Open Source Software 8(83):4748, 2023.
doi:10.21105/joss.04748

[SSC+22] S. Soiland-Reyes, P. Sefton, M. Crosas, L. J. Castro, F. Coppens, J. M. Fernández, D. Garijo, B. Grüning, M. L. Rosa, S. Leo, E. Ó. Carragáin, M. Portier, A. Trisovic, RO-Crate Community, P. Groth, C. Goble. Packaging research artefacts with RO-Crate. Data Science 5(2):97–138, 2022.
doi:10.3233/DS-210053

[The] The HDF Group. Hierarchical Data Format, version 5.
doi:10.5281/zenodo.17808558

[The24] The Galaxy Community. The Galaxy platform for accessible, reproducible, and collaborative data analyses: 2024 update. Nucleic Acids Research 52(W1):W83–W94, 2024.
doi:10.1093/nar/gkae410

[VMT22] N. Vallet, D. Michonneau, S. Tournier. Toward practical transparent verifiable and long-term reproducible research using Guix. Scientific Data 9:597, 2022.
doi:10.1038/s41597-022-01720-9

[WAB+25] S. R. Wilkinson, M. Aloqalaa, K. Belhajjame, M. R. Crusoe, B. de Paula Kinoshita, L. Gadelha, D. Garijo, O. J. R. Gustafsson, N. Juty, S. Kanwal, F. Z. Khan, J. Köster, K. P. von Gehlen, L. Pouchard, R. K. Rannow, S. Soiland-Reyes, N. Soranzo, S. Sufi, Z. Sun, B. Vilne, M. A. Wouters, D. Yuen, C. Goble. Applying the FAIR Principles to computational workflows. Scientific Data 12:328, 2025.
doi:10.1038/s41597-025-04451-9

[WDA+16] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. B. da Silva Santos, P. E. Bourne, J. Bouwman, A. J. Brookes, T. Clark, M. Crosas, I. Dillo, O. Dumon, S. Edmunds, C. T. Evelo, R. Finkers, A. Gonzalez-Beltran, A. J. G. Gray, P. Groth, C. Goble, J. S. Grethe, J. Heringa, P. A. C. 't Hoen, R. Hooft, T. Kuhn, R. Kok, J. Kok, S. J. Lusher, M. E. Martone, A. Mons, A. L. Packer, B. Persson, P. Rocca-Serra, M. Roos, R. van Schaik, S.-A. Sansone, E. Schultes, T. Sengstag, T. Slater, G. Strawn, M. A. Swertz, M. Thompson, J. van der Lei, E. van Mulligen, J. Velterop, A. Waagmeester, P. Wittenburg, K. Wolstencroft, J. Zhao, B. Mons. The FAIR Guiding Principles for

scientific data management and stewardship. Scientific Data 3:160018, 2016. doi:10.1038/sdata.2016.18

[ZYBK23] H. Zhao, A. Yildiz, N. Bagherinejad, J. Kowalski. PSimPy: GP emulation-based sensitivity analysis, uncertainty quantification and calibration of landslide simulators. In 14th International Conference on Applications of Statistics and Probability in Civil Engineering (ICASP14). Dublin, Ireland, 2023. Available at http://hdl.handle.net/2262/103542.